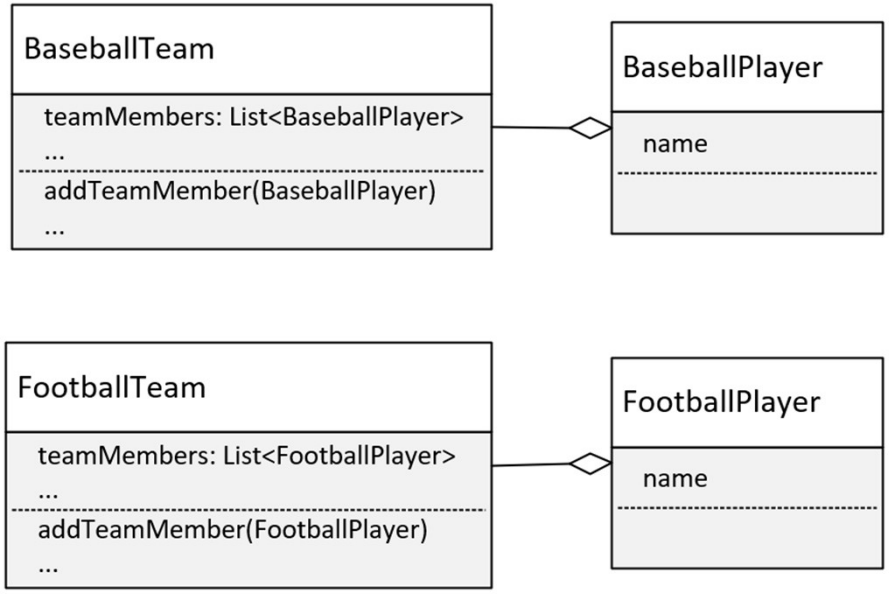
Solution 1: Duplicate code

We could copy and paste the BaseballTeam, and rename everything for FootballTeam, and create a FootballPlayer, as I'm showing on this slide.

This means you'd have to make sure any changes you made to one team or player, that made sense for the other team and player, had to be made in both sets of code.

This is rarely a recommended approach, unless team operations are significantly different.



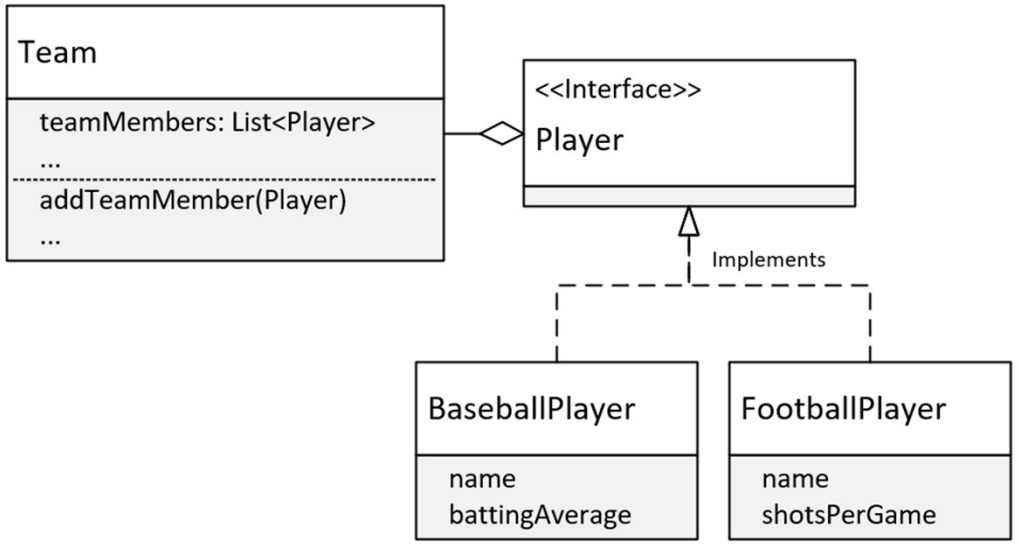
Solution 2: Use a Player Interface or abstract class to support different types of players

We could change Baseball team to simply Team, and use an interface type (or abstract or base class) called Player.

On this slide, I show a Team Class, and on this class, the members are a List of Players.

I've made Player an interface, and have BaseballPlayer and FootballPlayer implementing that interface.

This is a better design than the previous one, but it's still got problems.



Generic Type Parameters

I've already shown you that one way to declare a generic class, is to include a type parameter which I show here, in the angle brackets.

```
public class Team<T> {
```

Now, using T is just a convention, short for whatever type you want to use this Team class for.

But you can put anything you want in there.

Single letter types are the convention however, and they're a lot easier to spot in the class code, so let me encourage you to stick to this convention.

Generic Type Parameters

You can have more than one type parameter, so we could do T1, T2, T3.

```
public class Team<T1, T2, T3> {
```

But again convention says, that instead of using type parameters like this, it's easier to read the code with alternate letter selections.

And these are usually S, U, and V, in that order.

If we had three types, we'd probably want to declare this class as shown here, with T, S, and U.

```
public class Team<T, S, U> {
```

Generic Type Parameters

A few letters are reserved for special use cases.

The most commonly used type parameter identifiers are:

- E for Element (used extensively by the Java Collections Framework).
- K for Key (used for mapped types).
- N for Number.
- T for Type.
- V for Value.
- S, U, V etc. for 2nd, 3rd, 4th types.

Raw usage of generic classes.

When you use generic classes, either referencing them or instantiating them, it's definitely recommended that you include a type parameter.

But you can still use them without specifying one. This is called the **Raw Use** of the reference type.

The raw use of these classes is still available, for backwards compatibility, but it's discouraged for several reasons.

- Generics allow the compiler to do compile-time type checking, when adding and processing elements in the list.
- Generics simplify code, because we don't have to do our own type checking and casting, as we would, if the type of our elements was Object.